Module 4: Trusted Execution Environments

Task 3: Side-channel Attacks and Defenses [55 Points]

Prof. Shweta Shinde *shweta.shivajishinde@inf.ethz.ch*
Supraja Sridhara *supraja.sridhara@inf.ethz.ch*
Mark Kuhne *mark.kuhne@inf.ethz.ch*

Welcome to the fourth module of the Information Security Lab. This task is part of the Module-04 on *Trusted Execution Environments*, or TEEs in short.

1 Overview

This task is focused on studying information leakage via side-channels. The task is divided into 2 parts. In the first part, the goal is to study and analyze the sources of leakage and leverage them to extract program secrets. In the second part, the goal is to build defenses that prevent the leakage.

For the first part of this task (Tasks 3.1 and 3.2), you should think like an attacker. Here is the scenario: our victim program checks if an input matches a secret password. You want to extract the password but you cannot directly access it. The program does not have any logical or memory vulnerabilities. So the only way you can try to crack the password is by looking at the side-channel information. You will implement such password crackers.

In the second part of this task (Task 3.3), you should think like a defender. You will refactor an existing password checker implementation such that you prevent the leakage via side-channels.

For the first part of this task, we expect you to use Python v3.6.5. We will not accept submissions/solutions coded in any other language, including other versions of Python. *So please make sure that your submissions are compatible.* For the second part of the task, we expect you to submit a C file that implements the password checker.

Note that for all the tasks in this handout, address space layout randomization (ASLR) is turned off for the two VMs.

2 Attacks: Tasks 3.1 & 3.2

2.1 Getting Started

We have provided you with a VM image (`VM_Task3_1_2.ova`) which has several tools and infrastructure required for the task. Please boot the VM and get familiar with the following layout:

Listing 1: Directory structure of VM_Task3_1_2.ova. Note that, in this VM the task folders are named t1 and t2. This is intentional.

```
isl
├── t1/
│   ├── password_checker_1
│   ├── password_checker_1.c
│   ├── password.txt
│   ├── run.sh
│   ├── output/
│   └── samples/
└──  t2/
    ├── password_checker_2
    ├── password_checker_2.c
    ├── passwords
    ├── password.txt
    ├── output/
    └── run.sh
```

We strongly recommend that you use the default configuration (RAM, CPU and network) in the .ova file. This is important so as to ensure that the timing measurements you do locally (e.g., total execution time) will be the same on our grading VM. We will use the same VM configurations to grade your solutions.

2.2 Victim Program

For the first part of the assignment, we provide two victim programs and corresponding binaries.

`password_checker_1.c` & `password_checker_2.c`: These two source files are located in `/home/sgx/isl/t1` and `/home/sgx/isl/t2` folders respectively. These C programs compare a password guess (input string) with the secret password. They output whether the guess was correct or wrong. Please read the program source code carefully to understand the format for the inputs and the secrets. Assume that these programs are free of any memory vulnerabilities (e.g., buffer overflows).

`run.sh`: Script to statically compile the password checker programs to produce corresponding executable binaries. Note that the programs are statically linked to a math library (`-lm`), are compiled with optimizations turned off (`-O0`), and debug information enabled (`-g`).

`password_checker_1`, `password_checker_2`: Corresponding executable binaries produced using `run.sh`.

2.3 Secrets

The programs operate on a secret password.

`password.txt`: Both `password_checker_1` and `password_checker_2` load the secret password from this file in their respective folders. Thus, the secret is not embedded in the source code or the binary.

2.4 Tools

As an attacker, a few tools remain available at your disposal. For an assignment, it is challenging to setup real Intel SGX machines and ask students to extract side-channel information about enclaves. Instead, we have simplified it a little bit. We give you all the information that the attacker can observe using several sources of side-channels. Your task is to simply analyze this information and extract the secret. Below we describe how we collect this information.

`Pin`: is a dynamic binary instrumentation framework provided by Intel. The Pin framework provides a developer friendly way to write several dynamic program analysis tools. We will be using the Pin framework to mimic a real attacker that wants to extract the information about our victim program. The framework is already built and installed in the VM, you will not be required to change anything in the framework. The tool is located at `/home/sgx/pin-3.11-97998-g7ecce2dac-gcc-linux-master/`, the `pin` script runs the pin command.

`SGX_Trace`: This is a custom Pintool that we wrote on top of the Pin framework. When a program is executed with this Pintool, it records important information about the execution. Specifically, it records the address and opcode of each instruction that is executed in the program. Further, it also records the data addresses that were accessed (read/write) for each instruction. The Pintool is located in the `/home/sgx/pin-3.11-97998-g7ecce2dac-gcc-linux-master/source/tools/SGXTrace` folder, the `SGXTrace.cpp` file contains the Pintool source code. To run the SGXTrace tool, first enter the `/home/sgx/pin-3.11-97998-g7ecce2dac-gcc-linux-master/source/tools/SGXTrace` directory. Then run the following command:

```
../../../pin -t ./obj-intel64/SGXTrace.so -o <output-file>
-trace 1 -- <victim-program>
```

For example, you can trace the `ls /home` command as follows:

```
sgx@sgx-VirtualBox:
~/pin-3.11-97998-g7ecce2dac-gcc-linux-master/source/tools/SGXTrace$
../../../pin -t ./obj-intel64/SGXTrace.so -o ~/isl/ls.txt
-trace 1 -- ls /home
```

You will see the following output:

```
================================================
This application is instrumented by SGXTrace Pintool
PID: 7028
See file /home/sgx/isl/ls.txt for analysis results
================================================
sgx
```

`Trace Format`: The trace file (`/home/sgx/isl/ls.txt` in the above example) has a lot of information which may seem overwhelming in the beginning. Listing 2 shows an example trace snippet. Next, we explain the trace format:

Listing 2: Example trace snippet

```
1 E:0x7f71b3133102:C:7:add qword ptr [rax+0x8], rcx
2 R:0x7f71b3350e60:C
3 W:0x7f71b3fdea60:D
```

The first line is `E`, i.e. this line is about the current EIP, followed by the instruction address `0x7f71b3133102` in the virtual memory, followed by `C`, which means this is from the code section, and then the opcode 7, followed by the disassembly of the instruction `add qword ptr [rax+0x8], rcx`. Line 2 tells us that the instruction caused a read `R` at the address `0x7f71b3350e60` in the code section `C`. Line 3 tells us that the same instruction also caused a write `W` at the address `0x7f71b3fdea60` in the data section `D`. Please take a look at the binary assembly, input values, and corresponding trace entries to understand the details.

2.5 Trace Samples

We tested the victim program on a bunch of secret passwords. For each password, we made several guesses. We recorded all these executions using our SGX_Trace Pintool. So, we were able to simulate an attacker who doesn't know the password, but can make several guesses. For each guess, the attacker can observe an *execution trace* as collected by the SGX_Trace Pintool and the final output (either `correct` or `wrong`) printed on the console. We have provided several such trace samples for different passwords. The VM contains sample trace sets grouped by $\langle id \rangle$. Each folder named $\langle id \rangle$ contains traces for a particular secret password. For each secret password (e.g., `samples/`$\langle id \rangle$`/password.txt`), there are several traces over different guesses in the `/samples/`$\langle id \rangle$`/traces` folder (e.g., sample/$\langle id \rangle$/traces/magicbeans.txt, sample/$\langle id \rangle$/traces/magicbeant.txt). The `password.txt` file contains the secret password. Each trace file is named as the guess that was used to generate the trace. For example, the trace for the command `./password_checker_1 magicbeans` with secret password in `/sample/1/password.txt` is in file `sample/1/traces/magicbeans.txt`.

2.6 Goal

Given the same secrets, the victim program exhibits different execution behavior for different inputs. For instance, consider that the secret password is `abracadabra`. When we execute a password checker program with an input password guess of `guesspassword`, the program output will tell us that the input is incorrect. Thus, the output tells a little bit about the secret. In this case, it tells us that the secret password is not `guesspassword`. However, the attacker can try several inputs and observe other aspects of the execution. By analyzing how the program behavior changes with the change in the input, the attacker can extract more information about the secret.

In the case of the TEE threat model, the attacker can be the OS, the hypervisor, non-enclave programs, or malicious enclaves co-resident on the same SGX machine. As we will see in the class, this new threat model exposes several new avenues of information leakage, mainly via side-channels. In this part of the task, you are presented with a corpus of such information

that can be collected via side-channels in SGX. However, such low-level knowledge about a programs execution may or may not lead to a complete leakage of program secrets. The attacker has to find clever ways to make sense of the information collected via side-channel(s).

In this part of the task, we will learn how to analyze the information collected via side-channels, compare it across several executions over varying inputs, and then devise attack strategies. In the process, we will study the reduction in the attacker's uncertainty about the program secret.

2.7 Dos and Don'ts

Password Checker. Do not move, change, or modify the password checker source code or binary. Similarly, please do not change the provided sample execution traces and passwords. Any change will affect the program addresses and will not match the pre-recorded samples. You can examine the source code and the binary (say in read-only mode). Please be mindful of this especially when you test your solution on the traces or generate your own traces.

Pin & SGX_Trace Tool. We have provided these tools for your reference, in case you want to understand how the programs are being traced and what each line in the trace means. Please do not change these tools when you are coding or testing your solution. You can use the tools from Section 2.4 to trace some existing programs (e.g., `ls`) or test programs that you code to understand the details of the trace collection. That way, you can generate your own traces and get some insights into the sample traces that we have provided.

Trace & Trace Samples. To assess your solution for Task 3.1, we will first test it on the public sample traces provided with the VM. Then we will run it on a reserved test set with extensive sample traces (generated using other passwords) not provided with the VM. So make sure your solution computes the correct password at least on the sample traces. If you want to do more extensive testing, you can generate your own sample traces on passwords of your choice.

2.8 Coding Tasks

Input-output format. For Task 3.1 and Task 3.2, you should write python scripts named `submit_3_1.py` and `submit_3_2.py` respectively. The scripts should work with Python v3.6.5.

The `submit_3_1.py` should take 2 arguments; (a) arg1: $\langle /path/to/traces/folder \rangle$ (b) arg2: $\langle id \rangle$.

The `submit_3_2.py` should take only the $\langle id \rangle$ as an argument i.e., it only takes 1 argument. The scripts should create a file named `oput_`$\langle id \rangle$ in the `output` folder for the sub-task. Your script (`submit_3_1.py`) should write out the `password` for the folder of traces and a `status flag` (either `complete` or `partial`) to the output file. The `password` and `status flag` must be separated by a comma (`,`).

For example, on running the command `python3 submit_3_1.py /home/sgx/isl/t3_1/samples/1/traces 1`, the script should create a file named `oput_1`

in the /home/sgx/isl/t3_1/output folder. If you are sure that the complete password is magicbeans, your script should write magicbeans,complete to oput_1.

If you think that the given traces do not have sufficient information to recover the entire password—you were only able to recover a prefix, or you are not sure if you recovered all the characters in the password—your script can write the partial password that you were able to recover. For example, if you recover only partial password, say magic, then the oput_1 file should contain magic,partial.

If there are holes in your guessed password, you should place a single underscore (_) at the letter positions you were not able to recover (e.g., m_gic__ans). In this case, you have to use the partial flag. So, the oput_1 file should contain m_gic__ans,partial.

[5 points] Task 3.1:

Write a password cracking program that extracts the secret password by analyzing the execution traces of password_checker_1 binary provided to you. Please code in Python v3.6.5 as a single script file. Name your code script as submit_3_1.py.

Grading criteria:

- We will first evaluate your solution script on the public trace sets provided with the VM. If your solution script does not pass all public tests you get 0 points for Task 3.1. If your solution script passes all the public tests, we will evaluate it on the reserved trace set. Note that, you will not get any points for passing just the public tests.
- You will get 1 point for each sample trace set in our reserved traces (not provided with the VM) that your script writes the correct output for. There are 5 sets of reserved traces.
- You will get points for a trace set only if your script successfully writes out the correct password and status flag.
- Your solution script should only use the sample traces provided to you. Your solution for this task is not allowed to execute the password_checker_1 binary or the tracer.

[10 points] Task 3.2: Write a password cracking program that extracts the secret password for password_checker_2 binary by generating your own execution traces and then analyzing them. Please code in Python v3.6.5 as a single script file. Name your code script as submit_3_2.py.

Grading criteria:

- We will first evaluate your solution script on the public passwords (/home/sgx/isl/t1/samples/*/password.txt) provided with the VM. If your solution script does not pass for all passwords you get 0 points for Task 3.2. If your solution script passes all the public tests, we will evaluate it on the reserved password set. Note that, you will not get any points for passing just the public tests.

- You will get 2 points for each password in our reserved test set (not provided with the VM) that your script writes the correct output for. There are 5 reserved passwords.
- You are allowed to execute the `password_checker_2` binary and the tracer program in your solution any number of times in order to generate your own traces for your solution (this is not allowed for Task 3.1).
- For each set of sample traces, if your script fails to print the password, takes too long (more than 1 minute) to finish, or has errors in its execution, we will run it again for a total of 3 times. If in these 3 attempts it does not print the password, you will not get the points. Thus, it is important for your script to be reliable: remember the cracker should run outside GDB or any other tool that you may use during testing. You are responsible for error handling

2.9 Submission format

For Task 3.1 you should submit a Python file named `submit_3_1.py` on Moodle. For Task 3.2 you should submit a Python file named `submit_3_2.py` on Moodle.

2.10 Auto-grader

To evaluate your solution for Task 3.1, our grader program downloads a fresh copy of the VM image that was provided to you and boots it. Next, it copies your python script `submit_3_1.py` to `/home/sgx/isl/t1/` in the fresh VM it booted. For each set of traces, it executes `python3 submit_3_1.py` $\langle /path/to/traces/folder \rangle$ $\langle id \rangle$. Once the script is executed for all trace sets, both public and reserved, the grader program copies out the `oput_`$\langle id \rangle$ files from the `/home/sgx/isl/t3_1/outputs` folder to our machine. Finally, the files (`oput_`$\langle id \rangle$) are checked to evaluate your solution and calculate the final points of this task.

To evaluate your solution for Task 3.2, our grader program downloads a fresh copy of the VM image that was provided to you and boots it. Next, it copies your python script `submit_3_2.py` to `/home/sgx/isl/t2/` in the fresh VM it booted. For each password (public and reserved), it copies the password to `/home/sgx/isl/t2/password.txt` and then executes `python3 submit_3_2.py` $\langle id \rangle$ on the VM. Once the script is executed for all the passwords, both public and reserved, the grader program copies out the `oput_`$\langle id \rangle$ files from the `/home/sgx/isl/t2/outputs` folder to our machine. Finally, the files (`oput_`$\langle id \rangle$) are checked to evaluate your solution and calculate the final points of this task.

You can run this grader program on your system by downloading the grader-3-1_2.zip from Moodle. The grader program is distributed as a .zip file with the structure shown in Listing 3. Note that, the grader program provided to you only evaluates your solution on the public test sets. For the final grading your solution is evaluated on the reserved test sets in a manner similar to that on the public test sets.

Listing 3: Structure of grader-3-1_2.zip.

```
1 grader-3-1_2.zip
```

```
├──samples/chmod
├──grade_3_1.sh
├──grade_3_2.sh
├──check_oputs-3-1.py
├──check_oputs-3-2.py
└──README
```

Running the grader program yourself. Do not perform the following steps to run the grader program inside the task VM. You will have to use a Linux host machine with Ubuntu 20.04.3 LTS to run the grader program with VirtualBox installed.

To run the grader, extract the `grader-3-1_2.zip` to `grader-3` folder on your Linux host machine. Next, follow the steps below:

- execute `chmod +x <path to grader-3>/grade.sh` to give execute permissions for the script.
- For Task 3.1:
  - execute `<path to grader-3>/grade_3_1.sh <path to submit_3_1.py>`. The `grade_3_1.sh` script reads the path to the solution python file as the first command line argument.
  - execute `<path to grader-3>/check_oputs.py <eth-id>` that checks the `oput_`$\langle id \rangle$ and grades it. The final grade file `grade-3-1-<eth-id>` is created in the same directory. See Figure 1 for the final grade file format.
- For Task 3.2:
  - execute `<path to grader-3>/grade_3_2.sh <path to submit_3_2.py>`. The `grade_3_2.sh` script reads the path to the solution python file as the first command line argument.
  - execute `<path to grader-3>/check_oputs.py <eth-id>` that checks the `oput_`$\langle id \rangle$ and grades it. The final grade file `grade-3-2-<eth-id>` is created in the same directory. See Figure 2 for the final grade file format.

3 Defense: Task 3.3

3.1 Goal

For the second part of the lab, we will learn how to defend the vulnerable program against side-channel attacks. Specifically, we will consider the same adversary as in the first part who can glean information recorded in the traces. We will not specify the sources of leakage (e.g., caches, speculation pipelines, timing). Instead, we will capture the adversaries capabilities using the tracer. Although there are several approaches to harden the program against such a

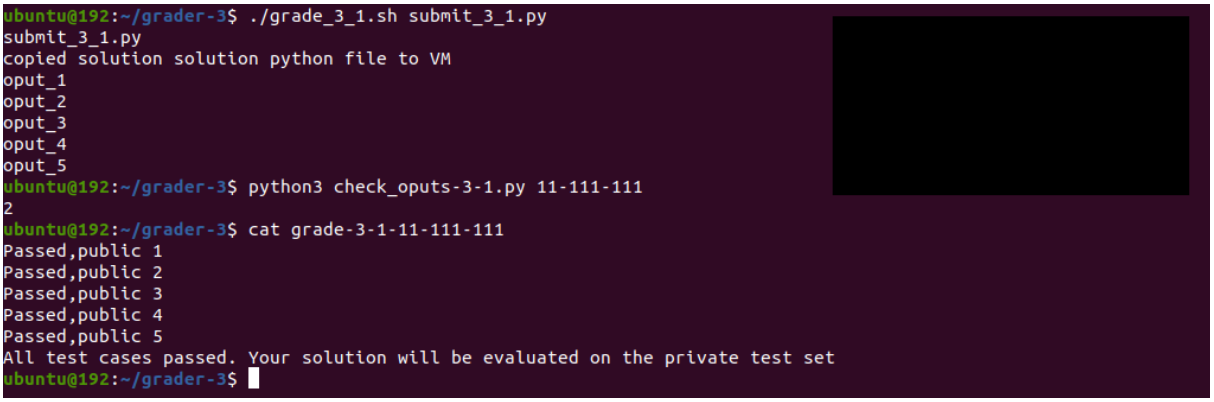

Figure 1: Example output of the grader for Task 3.1

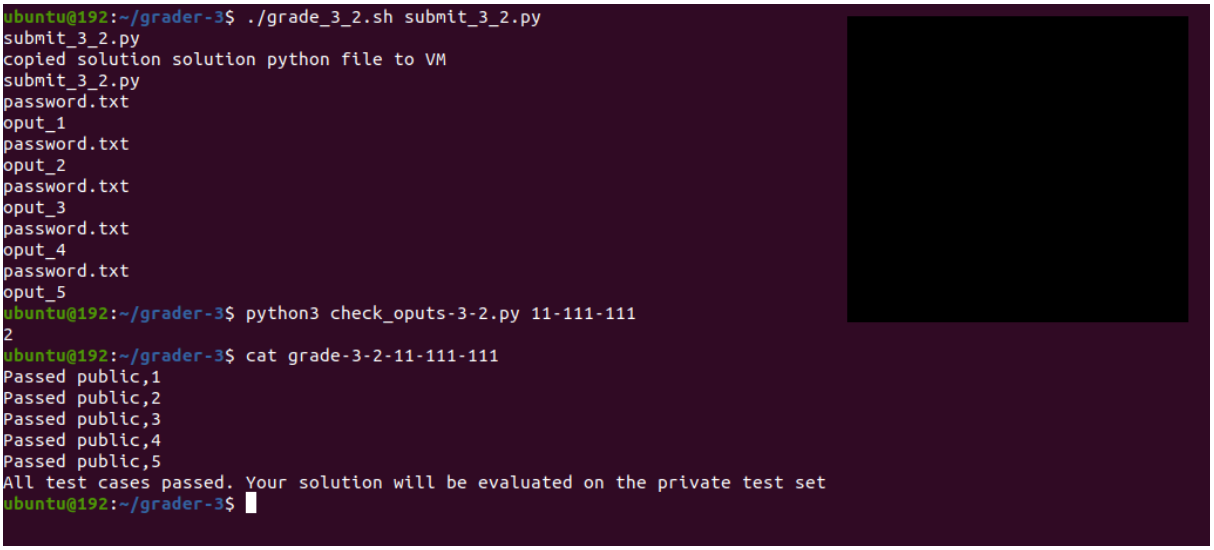

Figure 2: Example output of the grader for Task 3.2

strong adversary, for this task we will use a defense that makes the program execution deterministic. More precisely, if the program exhibits the same trace for all possible executions, the attacker will not learn any information about the inputs. Thus, in our defense we will enforce a deterministic execution that is independent of the inputs.

3.2 Target Program

For a concrete defense, we consider a simple password checker. This program is vulnerable to memory-access based side-channels, as you demonstrated with your attack. The functionality of the program is to read the correct password from a file and compare it with the guess password provided as a command line input. The program logic compares these two strings. If the strings match exactly (i.e., length and characters at each position), the program writes a bit 1 to the output file, whose name is the second command line argument. Otherwise, the program writes a bit 0 to the output file. See Figure 3 for the example execution and output.

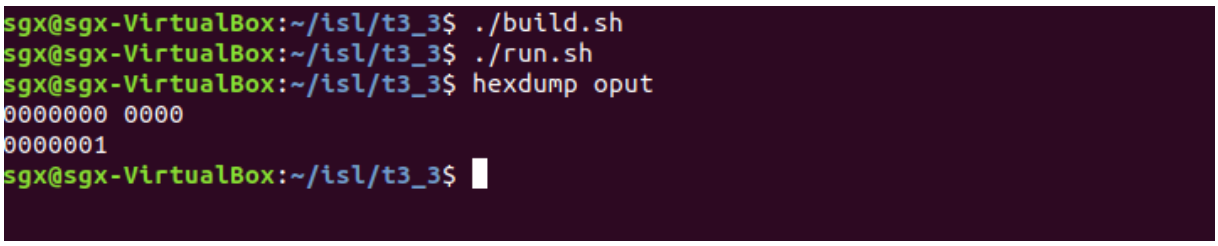

Figure 3: Example execution of password_checker_3.c

3.3 Input & Output Formats

For inputs, the program expects 2 command line arguments:

- arg 1: the $\langle guess \rangle$ password, length of 1 to 14 characters. Further, the characters are only lower-case characters a to z.
- arg 2: the name of the output file.

The program reads the $\langle correct \rangle$ password from the `/home/sgx/isl/t3_3/password.txt` file. The $\langle correct \rangle$ password can be 0 to 14 characters long. Further, the characters are only lower-case characters a to z.

The $\langle correct \rangle$ password in this file is padded with dollars ($) on both sides using the following scheme: the first $15 - l$ $s followed by l characters of the correct password, followed by $15 - l$ $s. For example, if the correct password is `magicbeans`, the file will contain `$$$$$magicbeans$$$$$`.

For outputs, the program creates a new file with a name specified by arg 2. The file is opened in binary mode (wb) and the program writes only one bit at the beginning of the file. At the end of the program execution, the output file size is 1 byte.

3.4 Getting started

Listing 4: Directory structure of VM_Task3_3.ova

```
isl
└── t3_3/
    ├── test/
    │   ├── test_setup.py
    │   ├── public_test_set.csv
    │   ├── run_tracer.py
    │   ├── diff_traces.py
    │   ├── traces/
    │   └── run_single.sh
    ├── dummy.c
    ├── password_checker_3.c
    ├── password.txt
    ├── build.sh
    └── run.sh
```

For this task please download the separate VM image VM_Task3_3 with the t3_3 folder. The tree structure is shown in Listing 4. Each directory and file in this tree has a specific purpose, as explained below:

- test/: This folder contains the public test set and scripts to run and analyze your programs, see Section 3.5 for more details.
- dummy.c: A dummy C program that is used to test your setup. See below for more details.
- password_checker_3.c: This file contains a simple password checker implementation that leaks information. You goal is to fix it.
- password.txt: this file contains the correct password for a single execution of your program. The password_checker_3.c reads this file for the correct password and compares it with the guess password.
- build.sh: this script builds the password_checker_3.c file and creates an executable binary named a.out in the same folder.
- run.sh: this script runs the /home/sgx/isl/t3_3/a.out binary, if it is present. Note that this script does not run the SGXTracer from Section 2.4. It takes 2 command line arguments: the $\langle guess \rangle$ password and the output file name.

We strongly recommend that you use the default configuration (RAM, CPU and network) in the .ova file. This is important so as to ensure that the timing measurements you do locally (e.g., total execution time) will be the same on our grading VM. We will use the same VM configurations to grade your solutions.

Testing your setup. To test that the tracer is installed correctly and computes the correct traces, run /home/sgx/isl/t3_3/test/test_setup.py. This script first compiles the dummy.c file to generate a binary a.out. Next, it runs the tracer from Section 2.4 on the a.out binary with different guess passwords of the same length as command line inputs. Finally, it tests that the generated traces are identical. If this test fails, it means that your environment is not correctly setup or your changes have corrupted the VM.

diff. In order to compare traces, we use a Linux utility called diff. The tool compares files and reports if they are identical or not. For the rest of the handout, we will refer to the action of using this tool as *computing the diff*.

3.5 Tools

To help you code this task, you can use the tools that you used for your attacks in Tasks 3.1 and 3.2. Specifically, the Pin framework and the SGXTracer explained in Section 2.4 can be useful.

Public Tests. To evaluate your solution, you can use the public tests in the `/home/sgx/isl/t3_3/tests/` folder. This folder contains the public test set and scripts to run the tracer and analyse the output traces. The files in the folder are as explained below:

- `public_test_set.csv` : This file contains comma-separated strings. The rows of the file are separated by newlines ('$\backslash n$'). The rows are of the form $\langle guess, correct, expected\_output \rangle$. The $expected\_output$ is 1 if the $guess$ password and the $correct$ password match, and 0 otherwise.
- `run_tracer.py` : Runs the `SGXTrace` from Section 2.4 on the full public test set using the `a.out` binary from `/home/sgx/isl/t3_3/`. Note that this script always passes `oput` as the second command line input to the `a.out` binary i.e., the name of the output file is always `oput`. It creates trace files in the `/home/sgx/isl/t3_3/tests/traces` folder. The names of the trace files of the form `trace_`$\langle guess \rangle$`_`$\langle correct \rangle$`.txt`. The script also creates a single `functionality.csv` file that contains rows of the form $\langle guess, correct, output \rangle$ by reading the binary file `/home/sgx/isl/t3_3/oput`.
- `diff_traces.py`: Computes the diff of all the traces in the public test set by executing `diff -q` $\langle file1 \rangle$ $\langle file2 \rangle$ and creates a file `traces_diff.csv` with rows of the form $\langle guess1, guess2, correct1, correct2, diff\_output \rangle$.
- `run_single.sh`: this script can be used to run the tracer on a single pair of $\langle guess \rangle$ password. The $\langle correct \rangle$ password is read from the `/home/sgx/isl/t3_3/password.txt` file. To run this script, execute `run_single.sh` $\langle guess \rangle$. This creates a trace file named `trace_`$\langle guess \rangle$`_`$\langle correct \rangle$`.txt` in the `/home/sgx/isl/t3_3/tests/traces` folder.

3.6 Coding Task

For this task your goal is to refactor `/home/sgx/isl/t3_3/password_checker_3.c` such that:

(a) the functionality is correct, i.e., the program computes the correct result for all possible guesses and passwords; and

(b) does not leak any information to the side-channel adversary described in Section 3.1.

To ensure (b), your program should exhibit deterministic executions for all possible $\langle guess \rangle$ and $\langle correct \rangle$ password pairs as discussed in Section 3.1. However, this can be challenging to do in the lab environment. First, we are limited to a specific configuration of VM, compiler, and user libraries. Second, the VMs have to run on a wide set of host hardware in student computers. Achieving determinism for these varying parameters is beyond the scope of this assignment. Therefore, for simplicity, we aim for a weaker definition of deterministic execution.

Concretely, the traces generated by `SGXTracer` in Section 2.4 should be the same for all pairs of $\langle guess \rangle$es of ***the same length*** and $\langle correct \rangle$ passwords of ***any length***. For example, traces for the pairs of guess and correct passwrords (`magic, magicbeans`), (`qwert,`

qwerty), and (pawnd, magic) should be identical because all guesses are 5 characters long. The traces for (qwerty, qwerty), (magicbeans, magic), and (justmagic, magicbeans) can be different because the length of the guesses differ.

To generate the traces you can use run_tracer.py, as explained in Section 3.5. Once the traces are generated, you can compute the diff of the traces using the diff_traces.sh script from Section 3.5. If your program exhibits a deterministic trace for all inputs, the $\langle diff\_output \rangle$ field in traces_diff.csv will be 0 for all rows.

While writing your solution program, please note the following:

- Always compile your program with the /home/sgx/isl/t3_3/build.sh script. It is important to use the same compiler flags. This means that you cannot use any libraries that are not already included in the compilation or change the compilation flags. When you compile your code, you should ensure that there are no errors or warnings. Our grader program will use this exact script. Your traces might vary if you use different compiler flags or compile manually from other paths. See Section 3.8 for more details.
- You should ensure that your program is functionally correct, compiles without errors or warnings, executes without runtime errors, and does not crash. We will not check for the memory safety of your programs (e.g., using a address sanitization library).
- Always use the /home/sgx/isl/t3_3/tests/run_single.sh or /home/sgx/isl/t3_3/tests/run_tracer.py scripts to generate traces for your program. These scripts set environment variables to specific values. Do not change the environment, if you do so the generated traces might differ even if your solution is correct.
- Your program should not directly access the tracer or attempt to corrupt it.
- Your programs must not have any persistent storage.

3.7 Submission format

For this task you should submit a file named password_checker_3.c on Moodle.

3.8 Auto-grader

To evaluate your program, our grader program downloads a fresh copy of the VM image that was provided to you and boots it. Next, it copies your program password_checker_3.c to /home/sgx/isl/t3_3/ in the fresh VM it booted. It then executes /home/sgx/isl/t3_3/build.sh on the VM to build the C file. The build generates the binary a.out in the same folder. Once the binary is created, the grader program executes

```
python3 /home/sgx/isl/t3_3/tests/run_tracer.py
```

that generates the traces as explained in Section 3.5. Finally, the grader program runs

```
/home/sgx/isl/t3_3/tests/diff_traces.sh
```

and copies out the traces_diff.csv and functionality.csv generated by the scripts to our machine. See Figure 4 for examples of the .csv files. These files are checked to evaluate

```
ubuntu@192:~/grader-3-3$ head functionality.csv
guess,correct1,output
wholetmein,wholetmein,1
letmein,wholetmein,0
guess,wholetmein,0
whoami,wholetmein,0
magicbeans,wholetmein,0
magic,wholetmein,0
qwerty,wholetmein,0
party,wholetmein,0
wholetmein,correct,0
ubuntu@192:~/grader-3-3$ head diff_traces.csv
guess1,guess2,correct1,correct2,diff_output
party,magic,wholetmein,wholetmein,0
party,party,wholetmein,magicbeans,0
party,magic,wholetmein,magicbeans,0
magic,party,wholetmein,wholetmein,0
magic,party,wholetmein,magicbeans,0
magic,magic,wholetmein,magicbeans,0
party,party,magicbeans,wholetmein,0
party,magic,magicbeans,wholetmein,0
party,magic,magicbeans,magicbeans,0
ubuntu@192:~/grader-3-3$
```

Figure 4: Examples contents of functionality.csv and diff_traces.csv

your solution and calculate the final points of this task. You can run this grader program on your system by downloading the grader-3-3.zip from Moodle. The grader program is distributed as a .zip file with the structure shown in Listing 5. Note that, the grader provided to you only evaluates your solution on the public test sets loaded from `public_test_set.csv`.For the final grading your solution we will load the test sets from a `private_test_set.csv`, instead of the `public_test_set.csv`. The rest of the grader will remain the same.

Listing 5: Structure of grader-3-3.zip

```
1 grader-3-3.zip
2 ├──grade.sh
3 ├──check_trace_outputs.py
4 └──README
```

Running the grader program yourself. Do not perform the following steps to run the grader program inside the task VM. You will have to use a Linux host machine with Ubuntu 20.04.3 LTS to run the grader program with VirtualBox installed.

To run the grader, extract the `grader-3-3.zip` to `grader-3-3` folder on your Linux host machine. Next, follow the steps below:

- execute `chmod +x <path to grader-3-3>/grade.sh` to give execute permissions for the script.
- execute `<path to grader-3>/grade.sh <path to password_checker_3.c>`. The `grade.sh` script reads the path to the solution C file as the first command line argument.
- execute `<path to grader-3-3>/check_trace_outputs.py <eth-id>` that checks the `traces_diff.csv` and grades it. The final grade file `grade-<eth-id>` is created in the same directory. See Figure 5 for an example execution of the grader program.

```
ubuntu@192:~/grader-3-3$ ./grade-3-3.sh password_checker_3.c &> out.txt
ubuntu@192:~/grader-3-3$ head out.txt
copied solution solution c file to VM
built binary
================================================
This application is instrumented by SGXTrace Pintool
PID: 3957
See file /home/sgx/isl/t3_3/test/traces/wholetmein_wholetmein.txt for analysis results
================================================
================================================
This application is instrumented by SGXTrace Pintool
PID: 3966
ubuntu@192:~/grader-3-3$ tail out.txt
================================================
================================================
This application is instrumented by SGXTrace Pintool
PID: 4164
See file /home/sgx/isl/t3_3/test/traces/party_magicbeans.txt for analysis results
================================================
The functionality is correct
ran tracer
run diff now
Traces matched for all guesses of same length
ubuntu@192:~/grader-3-3$ python check_outputs.py 11-111-111
2
ubuntu@192:~/grader-3-3$ cat grade-3-3-11-111-111
Public Tests, passed, 5 Pointsubuntu@192:~/grader-3-3$
```

Figure 5: Example execution of the grader program for Task 3.3

3.9 [40 Points] Grading Criteria

- 5 points if your solution passes all public tests
- We will test your program on the reserved test set only if it passed all public tests. The grading scheme for the reserved test set is:
  - 10 points if your program passes atleast $\frac{1}{3}^{rd}$ of the reserved test set.
  - An additional 10 points if your program passes $\frac{1}{3}^{rd}$ to $\frac{2}{3}^{rd}$ of the reserved test set.
  - An additional 15 points if your program passes all cases in the reserved test set.

We will only execute the binary generated from your submission file for a maximum of 5 second for each test pair of $\langle guess \rangle \langle correct \rangle$ passwords. After 5 seconds, our grader program will kill your process and read the `oput` file, if it was created by program. If the file was not created, the grader program will move on to the next test pair.

Please note that we will use a clean environment for each script. Our grading VM will have the exact same environment as the VM that we gave you. We will spawn a fresh VM to grade each submission. Do not assume that our grading machine will have:

1. any libraries or environment variables that are not available on the VM.
2. sudo or root permissions for running your script.
3. internet to connect to other machines.
4. open ports to launch and execute other scripts, daemons, or services.
5. persistent storage on our machine.

The `functionality.csv` and `traces_diff.csv` are auto-generated. Do not modify them. Any intentional attempts to write to these files will be detected and the grader program will terminate without evaluating your solution. Furthermore, any attempts to corrupt the machine our grader VM or host will result in zero points for this module.

4 Discussions

Friess Carl (carl.friess@inf.ethz.ch), Suter-Dörig Andris (sandris@student.ethz.ch), Mink Eric (minker@student.ethz.ch), Mark Kuhne (mark.kuhne@student.ethz.ch), Supraja Sridhara (supraja.sridhara@inf.ethz.ch) and Shweta Shinde (shweta.shivajishinde@inf.ethz.ch) will monitor the Moodle discussions. We encourage you to ask clarification questions on this public forum. That way everyone will benefit from clarifications we provide and it reduces duplicate questions. If you have a question that requires explaining your solution, please send a private email to all of us. Note that we will only answer questions that genuinely warrant a response. As a rule of thumb, we will not provide hints or help you to debug your code.
In conclusion, happy cracking & coding!